State of Indiana DMPP 2020 - Specifications
Message Header Processing in the Law Enforcement Environment using the Datamaxx Message Processing Protocol
(DMPP-2020)

Foreword

This document is meant to offer a brief discussion of the benefits of utilizing this protocol within Law Enforcement systems. This protocol has been chosen as the mechanism to provide compliance with national standards while also providing an environment that is not burdensome for regional/remote systems to implement.

Control Terminal Agencies (CTA) within each State must deploy systems that comply with the requirements of national agencies such as NCIC and NLETS. Some of these requirements are listed below:

Implementing a protocol that provides guaranteed message delivery

Identification of users responsible for transactions

Ability to process transactions that meet national standards

Ability to process image data

All of these requirements are implemented in the system with this protocol. DMPP 2020 is an application to application protocol which guarantees delivery of messages. The CTA must be able to determine the individual operator responsible for any message that traverses through the system. The OpenFox™ system as implemented in conjunction with the DMPP 2020 protocol and the inclusion of the user-id field achieves this requirement. The system supports all standard NLETS and NCIC 2000 transactions. In addition, the system using will also be able to process images. DMPP 2020 is one of the solutions that provides the framework required to process binary data.

In addition to the national standards the CTA has an obligation to provide an environment to regional systems which is consistent and can be implemented with a reasonable amount of effort while using state-of-the-art protocols and network facilities. The DMPP-2020 protocol as implemented in the OpenFox™ system provides such an environment.

The DMP2020 portion of this document was written by Datamaxx Applied Technologies and describes the protocol. The remainder of the document has been written by Computer Projects of Illinois, Inc., and is meant to offer further details and message format examples. This document will be updated at a later date with specific examples for your implementation.

MESSAGE HEADER PROCESSING IN THE LAW ENFORCEMENT ENVIRONMENT
USING THE DATAMAXX MESSAGE PROCESSING PROTOCOL®
DMPP-2020®
Technical Specification

Published By:

DATAMAXX APPLIED TECHNOLOGIES, INC.
3780 PEDDIE DRIVE
TALLAHASSEE, FLORIDA, 32303
(850) 575-1023

http://www.datamaxx.com Revision Levels:

Revision 0, August 1996
Revision 1, October 1996
Revision 2, June 1997
Revision 3, September 1997 (Registered Copyright – TX 4-624-223)
Revision 4, September 1997
Revision 5, September 1998
Revision 6, July 2000

Introduction.

The purpose of this paper is to define a specification that can be implemented to provide robust message handling in the Law Enforcement Environment.

As the transition to modern communications protocols continues, new problems and challenges are presented to developers. This is especially true with "Open Systems", in

which there are components from various vendors, all of which must operate in harmony.

With legacy systems, one vendor had control of processing, from the end user keyboard to the host system and thus could control all standards, and could implement necessary functionality to ensure that all messages were delivered reliably.

With "Open Systems" and diverse vendors, functionality tends to be implemented as a series of layers, with information being passed up and down between layers. Complicating this is that fact that the layers may be implemented as a series of disparate free running processes, in which data is passed back and forth. Thus, an application may send data through several layers and processes about which it has no knowledge. Each process or layer may acknowledge to the previous process or layer that the data was successfully processed -- however error messages are often not communicated to previous processes, the chain, and thus the originating application may not be aware of an error. Thus the need for "application to application" or "end to end" acknowledgment.

Complicating the situation is that "Open Systems" are truly open, as they are designed to allow easy interconnection. This immediately provides points of access that can be used for unauthorized or abusive use of a system.

A further factor is that new protocols are "peer to peer" and do not provide a continuous status monitoring (as is the case with "master slave" type protocols). This can lead to situations in which an application can send a message to a destination that can not process it. Since there is no immediately available status, error indications may not be provided for several minutes (or at all) and the sending application will not be aware of the situation.

Consider the following scenario:

Host prepares a message for transmission.

Host passes the messages to Communications sub-system.

Communications sub-system passes message to communications controller.

Sub-system sends message immediately to destination, but is not aware if any intermediate devices (e.g. bridges or routers) are inoperative.

Remote communications processor receives message, acknowledges it and places it in a buffer.

Remote application crashes before reading buffer, or operator powers system off.

In this scenario the host application would consider that the messages has been correctly processed, when indeed it was not. Furthermore, many messages may have been sent and buffered for a remote application that never processes them. There are also many other potential points of failure that can leave the host in a state assuming a message was delivered, when it was not actually delivered.

In order to eliminate these potential points of failure, a structure must be defined that can be used universally. The approach defined herein uses a "Message Header" processing to achieve full end to end confirmation of all messages.

The processing strategy is known as the "Datamaxx Message Processing Protocol (DMPP-2020®)"

Concepts

In developing the message header processing, many factors were considered. These include:

Compatibility with NCIC designs Compatibility with State designs Full message delivery confirmation Communications Protocol Independent Applicable to all processing platforms Programmer friendly Support for security issues Support for data encryption Features can be configured to meet different requirements Flow control is automatically provided to avoid flooding of a target system

The design that evolved, after much research, involves the implementation of a special header in each message packet. This header contains control fields that can be used to provide all functionality, as needed. The header can also be defined as "optional" in order to allow remote systems to be converted as available, rather than requiring a "big bang" conversion.

This header will be referred to as the "Extended Message Header" throughout this paper.

A discussion of each of the concepts is warranted, in order to provide background and rational for the design.

Compatibility with NCIC designs

This design leverages off the structure proposed for the NCIC-2000 system, in order to reduce research and development time. It is not though, an exact copy of the NCIC structure.

Compatibility with State designs

This design allows the Extended Message Header to be placed in front of existing message formats, with no requirement to change those formats. This alleviates the requirement to modify existing processing applications.

Full Message Delivery Confirmation

The Extended Message Header provide both positive and negative confirmation of message delivery. For negative delivery confirmation a reason code is provided.

Communications Protocol Independent

Although the obvious protocol that this specification can be applied to is TCP/IP, it is actually protocol independent. It can operate on any binary transparent protocol, ranging from serial links (e.g. mobile communications via CPDP IP packets) to mainframe protocols (e.g., LU 6.2).

Applicable to all Processing Platforms

This design is compatible with all processing platforms. This is achieved by careful sizing and alignment of all data fields, in order to avoid alignment and size specification errors that are generated by some processors.

Programmer Friendly

The design guards against assumptions made by various compilers. For example, some compilers will automatically initialize data structures to null values, or just plain junk. This can lead to subtle processing flaws. Thus this specification does not allow any command, directive or response code that is all null values, and requires that all values be verified. It is also programming language independent. All Extended Message Header processing is symmetrical with respect to direction (inbound and outbound).

Support for Security Issues

The Extended Message Header provides for full authentication of all connections, including dynamic re-verification of connections at random intervals.

Support for Data Encryption

The Extended Message Header provides for full encryption of the data portion of messages. This allows a full software solution to be implemented, independent of all communications hardware. Dynamic key update and control is supported.

Features can be Configured to Meet Different Requirements

The features can be configured to meet the needs of a specific system. For example, the Extended Message Header can be implemented using a few of it capabilities and then more features can be activated as required.

Levels of Implementation

The specification can be implemented as "levels of service", depending on what options are selected. Thus it can be adapted to many different needs and environments.

Flow Control

The Extended Message Header can provide a natural flow control, if desired by the implementer.

Extended Message Header

The Extended Message Header is a structure that is inserted in a cleanly delineated message block. The general structure of the message block is as follows:

STAP	4 character start pattern
Block Length	32 bit signed integer (See note below)
Header	Extended Message Header (Defined in Section 4)
Data	Variable length data
STOP	4 character stop pattern

Note: The Block Length field encompasses the whole packet, including the STAP (Start Pattern), Block Length field itself, Extended Message Header, data (if any present) and the STOP (Stop Pattern).

For consistency across platforms, all values in the header are stored in "Network Byte Order". This order places the most significant byte first, descending to the least significant byte reading to the right. This is contrary to method used on some Intel platforms (notably the 80X6 family) and thus the implementation must handle this situation as required.

Definitions

The "STAP" (Start Pattern) and "STOP" (Stop Pattern) are currently defined as hexadecimal patterns as follows:

STAP	ff,00,aa,55	(\xff\x00\xaa\x55)
STOP	55,aa,00,ff	(\x55,\xaa,\x00,\xff)

The minimum block size is 28 characters, which can occur when the Extended Message Header length is 16 and there is no data present. The maximum block size is 2,147,483,647 (231 – 1). Thus, the value of the Block Length field must never be less than 28 or more than 2,147,483,647.

Extended Message Header Format

The Extended Header Message has the following required format:

Header Length- 16 bit signed integer Function Code - 16 bit signed integer
Validation Field - 32 bit unsigned integer Data Length - 32 bit signed integer Status Code - 16 bit signed integer Destination - 16 bit signed integer

The Extended Header Message has the following optional extension for encryption:

Length - 16 bit signed integer
Request Type - 16 bit signed integer
Key Id - 32 bit unsigned integer

In the following charts, all numbers are expressed as decimal integers. They can be converted to other number systems (e.g., octal or hex) as required. Note how the use of zeros is consistently avoided.

Each field is discussed in detail, as follows:

Header Length

The length is a 16 bit integer that encompasses all the header data, including the length field. It will be either 16 or 24, depending on whether or not an encryption control is present.

Function Code

The Function Code defines the processing path of the message. Currently defined values include:

1 - Data message with no acknowledgment, final block
2 - Data message with acknowledgment, final block

3 - Data message with no acknowledgment, more blocks to follow (See note below)
4 - Data message with acknowledgment, more blocks to follow (See note below)
17 - Positive acknowledgment to data message (Status Code is set to "Successful receipt of data message")
18 - Negative acknowledgment to data message (Error is defined in the Status Code field)
33 - Request status of system
34 - Response to status request (Status is defined in the Status Code field)
49 - Send Coded Message 1
50 - Send Coded Message 2
65 - Positive response to Coded Message 1
66 - Positive response to Coded Message 2

Note: Function Codes 3 and 4 are used to indicate that the message will be sent in multiple blocks with Function Codes 1 and 2 used to indicate the last block. Each block in such messages must use successive values in the validation field.

Validation Field

This unsigned integer field defines a number that is used to create a unique identification for each message, and will be returned on its corresponding acknowledgment. Its format is up to the implementer. This value may be all zeros, as it is not inspected but simply returned to the requester intact.

Data Length

This field defines the length of the actual data portion of the message. It is used for redundancy checking. It must be zero for status and status response messages. The maximum value is 2,147,483,619 (231 – 1 – 28).

Status Codes for Request Messages

This field contains the status code that can be included in request messages. Currently defined values include:

01 - Message may contain binary object in Unisys format
02 - Message doesn’t contain binary object
03 - Message contains binary object in NCIC transaction format
04 - Message contains binary object in NCIC response format
05 - Message contains binary object in DSEO-2020 format
06 - Message contains binary object in Unisys format
33 - Message may contain binary object in Unisys format

Note: Any message that can contain a binary object in any of the supported formats can contain multiple binary objects but they must all be in the same format.

Status Codes for Response Messages

This field contains the status code that can be returned in responses. They should be used only with responses -- never part of request messages (i.e., status codes are not "piggybacked" onto a request).

The code returned will depend on the type of request received, (e.g., a write request with acknowledgment, or an explicit request for status). Currently defined values include:

01 - Successful receipt of data message
17 - Permanent (i.e., non recoverable) error occurred (e.g., disk failure)
18 - Temporary (i.e., recoverable) error occurred (e.g., printer out of paper)
19 - Logical error occurred (e.g., too many messages received too quickly, and thus a queue containing acknowledgments filled up)
20 - Message length exceeds maximum, message will be discarded
33 - Queried destination is available and ready
34 - Queried destination is available, but not ready (e.g., printer has buffer space, but is out of paper)
35 - Queried destination is not available and not ready
49 - Invalid function code received
50 - Invalid (or non-existent) destination received
51 - Invalid Extended Message Header format (or length) received
52 - Function not supported
65 - Attempt to start encryption with no key definition
66 - Invalid encryption header format (or length) received
67 - Encryption not supported.

Destination

This 16 bit integer defines a logical destination. This permits a packet to be addressed to different logical units, and effectively creates a "cluster" at a location. The actual definition is up to the implementer and the configuration. This permits logical units to be defined for specific purposes (e.g., a destination for "Hit Confirmation" messages), and permits implementation of message priorities. The value of 0 is invalid. The value of "-1" is considered a broadcast to all defined destinations.

Encryption Header Length

This 16 bit integer defines the length of the optional encryption header. A length of zero is invalid.

Encryption Request Type

This 16 bit integer defines the encryption function requested.

Currently defined values include:

1	-	Start encrypting messages
2	-	Stop encrypting messages
17	-	Load encryption Key
18	-	Clear encryption key
33	-	Set key identification

Key Identification

This 32 bit integer defines the index into the key table to locate the key to be used for encrypting future messages.

Service Levels

The DMPP-2020™ specification allows for service levels. A service level defines that functionality that has been activated for a given endpoint on a communications network.

The following service levels are defined:

Level 1 provides the functionality for handling message header functions from 1 through 47 (as they may be defined). This functionality encompasses guaranteed delivery of messages and full status checking, but does not include authentication or encryption.

Level 2 provides the functionality as described in Level 1 and adds the functionality for system authentication (function codes 49 through 79, as they may be defined).

Level 3 provides the functionality as described in Level 2 plus adds the encryption options via the extensions for encryption.

Implementation Notes

The following notes are presented to give an insight into how the Extended Message Header may be applied to various functions.

Integer Values

In this specification all integers are positive signed values, unless otherwise noted.

Destination Codes

The destination field does not have to replace existing header structures. It is meant to augment them. This technique permits many logical units to be addressed by a single Host address (e.g., a single TCP/IP address). This eliminates large control tables, and their associated maintenance (e.g., holes in firewalls). The application may still process existing headers (e.g., those used on a BiSync 2780 line).

Flow Control

By use of the "Write with Acknowledge" function, flow control may be achieved. The application can be structured to allow any number of messages to be outstanding at any time, subject only to the limits of the receiver. If the limit is set to 1, automatic flow control is achieved.

Keep Alive Timer

This implementation provides full keepalive support, at the application level. A keepalive probe is a packet with a Request Status Function code and no data length. If an appropriate Response to Status Request is returned, then the connection is intact. Note that this can also be used to temporarily suspend traffic by responding with a Status Code 34 (temporarily unavailable).

Coded Messages

Coded messages are used to authenticate connections. Their use is specific, as follows:

A Session requesting a connection provides a predictive string of data (e.g., a logical name) and encodes it in such a way that the receiver can decode it. This can be done by using a known element (e.g., System Name, Date, circuit number, telephone number, etc.) and encoding it using a Huffman coding, or other encoding process. It sends it as a Coded Message 1 Function to the receiver.

The receiving session encodes a similar string (that is why it must be predictive) and compares it to the received string. If a match is found, a response code of 65 is sent, with no data. If no match, the receiver is silent (Why tell the crook how he failed).

Either side of the session may send a Coded Message 2 request at any time. The Coded Message 2 has a random data string as its data portion. The receiver then adds another predictive string of data to the coded data, re-encodes it and returns it as a response of code 66 to the sender.

The sender of the Coded Message 2 analyzes the response. If valid, processing continues (there is no response). If invalid, the connection is terminated, due to suspected invasion of the system.

The exchange of Coded Message 2 functions may occur at any time, thus creating a" keep-alive", as well as continually re-authenticating connections.

The encoded data in the Coded Message 2 may also be used as the encryption key, by inserting the optional encryption header.

Encryption Functions

The encryption functions are implemented implicitly. The presence of the optional encryption header defines an encryption function. If a Write Data function is performed with the encryption header, and the header defines a key load, then the data portion of the message is assumed to be the new key. This is consistent with the concept of loading the Coded Message 2 data as the key.

The encryption header will only require that the Key Identification field be present for the "Set Key Identification" function. It will be ignored for other functions. The length field must always be correct, though.

The "Set Key Identification" is used for systems that do not want to exchange actual keys as data, but prefer to keep a table loaded at a site. In that case, the key id is the index into the table.

Note that this does not speak to the encryption algorithm actually used. The algorithm strategy must be defined by the implementer.

Configuration Control

The features listed may be made configurable. For example, some systems may not support encryption, while others may allow many messages to be queued before acknowledgment. Other systems may require coded messages. These should all be implemented via service levels, not by specific option enabling techniques.

Precise Error and Status Reporting

The response codes permit isolation of errors clearly and cleanly. For example, there are codes for both "Invalid Function" and "Unsupported Function". This permits an interface to query a peer interface to determine what level of functionality is supported.

Current OpenFox™ Implementation

This section of the specifications is meant to provide practical examples for the implementation of the DMPP-2020 protocol in the OpenFox™ environment. The

following section will document the specific technique which OpenFox™ uses to provide reliable, binary object capable communications. The OpenFox™ system embraces the widely accepted standards of communication put forth in the NCIC and NLETS TCP/IP specifications and therefore implements DMPP-2020 in a manner complying with these national standards. The OpenFox™ system currently implements DMPP-2020 service level 1. The system uses the application level acknowledgments to reliably deliver messages, as well as the status checking function to implement an idle line timer. The OpenFox™ system requires the segmentation of large messages and an indication of which, if any, message segments contain image data. The OpenFox™ system will require client data messages to present images in the DSEO-2020 format. The OpenFox™ system does not use the authentication or encryption functions specified in service levels 2 and 3 at this time.

Message Header Fields

There are six fields in the extended message header, which are all used by OpenFox™. The fields, and the appropriate values, appear below.

Header Length This field is always set to 16 (hex 0010).
Function The functions supported are: (hex)
0001 - Data Message, no ACK, Final Block
0002 - Data Message, ACK requested, Final Block
0003 - Data Message, no ACK, More to Follow
0004 - Data Message, ACK requested, More to Follow
0011 - Positive ACK to data message
0012 - Negative ACK to data message
0021 - Request system status
0022 - Status response
Validation The contents of this field are returned by OpenFox™.
Data Length This field represents the data length as an unsigned 32-bit number. Please note that no single block may be larger than 65,535.
Status The OpenFox™ uses this field as documented in the DMPP-2020 spec.
Destination The value is always set to hex 0001 on outgoing messages, and ignored on inbound messages.

Keep Alive

The OpenFox™ uses the status request/response function to act as an idle line timer. OpenFox™ will terminate a connection that has had no activity for 60 seconds. To prevent an idle connection from terminating, clients are expected to issue a system status request message before 60 seconds of idle time. CPI recommends sending this request every 45 seconds if no other traffic has been sent during that time. The OpenFox™ will respond with a ‘system available’ status response and reset the idle timer for the connection.

Message Segmentation

To maximize resource efficiency at the central site and to manage a large number of client connections the OpenFox™ system requires that messages larger than 65,535 bytes be segmented. The term segment and block are used interchangeably. If a DSEO-2020 object is present in a message, it must be completely contained within a single message segment. Please note that a single message segment may contain multiple DSEO-2020 objects (so long as their combined size is under the 65,535 byte limit). A message may be broken into any number of segments, and each segment need not attain the 65,535 byte maximum. If the function code for a data message requests an ACK, and is not the final block, the next block should not be sent until the ACK for the prior block is received. Likewise, after sending a final block requesting an ACK, the next message should not be started until the ACK is received. If no ACK is received for a data block within 60 seconds the connection should be closed and a new connection attempted. Any partially completed message (some blocks sent and ACK'ed but not all) should be resent in its entirety upon successful establishment of the new connection.

Binary Objects

As documented above, OpenFox™ requires that all inbound and outbound objects be wrapped in DSEO-2020 format. An object when present must be completely contained with a single segment. The status field in the DMPP-2020 header should reflect the content of the block. The two status codes used are:

01 Message segment contains no object data
05 Message segment contains at least one DSEO-2020 formatted image

Please note that in the DMPP-2020 specification status code 01 states "Message may contain binary object in Unisys format". Since OpenFox™ does not support Unisys formatted objects this code is used to indicate no object is present. OpenFox™ will only scan segments for DSEO-2020 objects if they have the status code set to 05. OpenFox™ will insure that all segments bound for the peer have the status code set correctly, so the peer need not scan for DSEO-2020 objects if the segment status code is set to 01.

Message Examples

The following are example messages taken from a live system TCP/IP trace. First, we'll look at a status request message. The OpenFox™ received the following message from a client device:

```
Offset      Hex Data                          ASCII Equivalent
--------    --------------------------------  -----------------
00000000    ff00aa55 0000001c 00100021 31363138    |...U.......!1618|
00000010    00000000 00210001 55aa00ff             |.....!..U...    |
```

The message breakdown is:

FF00AA55	Start Pattern
0000001C	Message length (total length of this data message)
0010	Extended Header length (always 16 - hex 10)
0021	Function - Request system status
31363138	Validation Code - this will be returned (see response below)
00000000	Data Length - this is zero for status messages
0021	Status Code - ignored
0001	Destination - ignored
55AA00FF	Stop Pattern

This message caused OpenFox™ to reset the idle timer for this connection, and respond with the following message:

```
Offset      Hex Data                          ASCII Equivalent
--------    --------------------------------  -----------------
00000000    ff00aa55 0000001c 00100022 31363138    |...U.......'1618|
00000010    00000000 00210001 55aa00ff             |.....!..U...    |
```

The message breakdown is:

FF00AA55	Start Pattern
0000001C	Message length (total length of this data message)
0010	Extended Header length
0022	Function - Status Response
31363138	Validation Code - echoed from the status request
00000000	Data Length - this is zero for status messages
0021	Status Code - Available and ready
0001	Destination - always set to 1
55AA00FF	Stop Pattern

The next examples are a data message received from the client, and the ack returned by the OpenFox™. First, the following message is received by OpenFox™ (from a client device):

```
Offset      Hex Data                          ASCII Equivalent
--------    ----------------------------------    ---------------
00000000    ff00aa55 00000053 00100002 32313230    |...U...S....2120|
00000010    00000037 00210001 464f5859 2e52512e    |...7.!..FOXY.RQ.|
00000020    2a484152 52592e4f 4b4f4850 30303339    |*HARRY.INOHP0039|
00000030    2e53442e 4c49432f 33413236 3739362e    |.SD.LIC/3A26796.|
00000040    4c49592f 31393939 2e4c4954 2f504355    |LIY/1999.LIT/PCU|
00000050    aa00ff                          |...          |
```

The message breakdown is:

```
FF00AA55            Start Pattern
00000053            Message Length
0010            Extended Header length
0002            Function - Data message, ACK requested, final block.
32313230            Validation Code
00000037            Data length - length of the actual message data (from
FOXY to LIT/PC).
0021            Status Code - ignored
0001            Destination - ignored
464F thru 5043      Message Data (text)
55AA00FF            Stop Pattern
```

The OpenFox™ responds with:

```
Offset      Hex Data                          ASCII Equivalent
--------    ----------------------------------    -----------------
00000000    ff00aa55 0000001c 00100011 32313230    |...U........2120|
00000010    00000000 00010001 55aa00ff            |........U...    |
```

The message breakdown is:

FF00AA55	Start Pattern
0000001C	Message Length
0010	Extended Header length
0011	Function - Positive ACK
32313230	Validation Code - echoed from the input message
00000000	Data length - zero
0001	Status Code (meaningless)
0001	Destination (always 1)
55AA00FF	Stop Pattern

The next two examples are a message generated by OpenFox™ and sent to a client, as well as the client's response. First, the following message was sent by OpenFox™ to a client device:

```
Offset      Hex Data                          ASCII Equivalent
--------    ---------------------------------    -----------------
00000000    ff00aa55 000000b4 00100002 00000001    |...U.............|
00000010    00000098 00010001 464f5859 2e2a4841    |........FOXY.*HA|
00000020    5252592e 4e434943 20202020 20202031    |RRY.NCIC       1|
00000030    33363731 2031373a 33353a32 32204d52    |3671 17:35:22 MR|
00000040    49203039 30313831 0d0a464f 58592020    |I 090181..FOXY  |
00000050    20202020 20303030 30392031 373a3335    |     00009 17:35|
00000060    3a323220 30362f31 362f3230 30300d0a    |:22 06/16/2000..|
00000070    0d0a314c 3031464f 58592c4d 52494430    |..1L01FOXY,MRID0|
00000080    39303138 300d0a30 4b304850 30303339    |90180..0K0HP0039|
00000090    0d0a4e4f 20524543 4f524420 4c49432f    |..NO RECORD LIC/|
000000A0    33413236 37393620 4c49532f 53440d0a    |3A26796 LIS/SD..|
000000B0    55aa00ff                              |U...            |
```

The message breakdown is:

FF00AA55	Start Pattern
000000B4	Message Length
0010	Extended Header length
0002	Function - Data message, ACK requested, final block
00000001	Validation Code - should be returned in the client's ACK
00000098	Data Length - from FOXY to 'CR''LF'

0001	Status Code (meaningless)
0001	Destination (always 1)
464F thru 0D0A	Message Data
55AA00FF	Stop Pattern

The client responded with the following acknowledgment:

```
Offset      Hex Data                          ASCII Equivalent
--------    ----------------------------------    -----------------
00000000    ff00aa55 0000001c 00100011 00000001    |...U.............|
00000010    00000000 00010001 55aa00ff             |........U...    |
```

The message breakdown is:

FF00AA55	Start Pattern
0000001C	Message Length
0010	Extended Header length
0011	Function - Positive ACK
00000001	Validation Code - returned from the OpenFox™ output
00000000	Data Length - zero
0001	Status Code - ignored
0001	Destination - ignored
55AA00FF	Stop Pattern

The examples represent the normal operation of a line. The two systems exchange data messages, and during idle periods the first example of status request/response is conducted.

Message ACKs

One final consideration that is not covered in the examples is that data messages may be sent with a function code of "0001 - data message, no ACK, final block". If the OpenFox™ sets this value in the function field, it is not expecting an ACK to the message, and one should not be sent to OpenFox™. Likewise, if a client device sets this value in the function field, OpenFox™ will honor it and not send an ACK. CPI recommends the use of the message ACKs for all standard user transactions.

Operator Identification

In order to comply with the CJIS Security Policy published by the FBI, the OpenFox™ system supports the identification of device operators. This feature is implemented through the use of the message header from the remote system.

The system will validate the User ID and report a security violation if the User ID has not been configured or the user-id is not associated with the incoming station.

Input Message Formats

Messages from the trusted server workstation to the OpenFox™ system will be constructed depending on the destination. For this particular type of service the OpenFox™ system will be configured to support a communications interface called the 'server' and either one or all nodes behind the interface called 'TDAC' (Trusted Destination Address Code).

The header associated with input from these devices will be as follows:

TDAC.Reference.UserID.

Where:

TDAC	Is the name of the device from which the message is originating
Reference	Is a 10 character alphanumeric field
UserID	The user ID field

Message input from these devices will follow the NLETS Format and the NCIC 2000 Format as described below.

The NLETS Format

These messages include all of the valid NLETS MKE's, as well as any State specific MKE's for which the processing rules are identical to those of NLETS. They will follow the NLETS standard message format except that the TXT statement will be optional. The input message must contain a list of the detinations in order for the system to route the messages.

Following is a description of the NLETS format:

[header.]MKE.ORI.Destinations.Control-Field.TXT(optional) and text.

Where:

MKE The MKE field is a valid NLETS message key must be configured in the system, if not an error message will be returned to the input station indicating 'invalid MKE'. In addition, the incoming session will be checked to see if it has the authority to execute the message key. If not an error message will be generated back to originating station indicating 'not authorized to use MKE'.

ORI The ORI must be the valid State ORI assigned to the input station. If the ORI is not valid or if it is not assigned to the originator an error will be returned.

DEST The destination field is variable in length, it must be composed of a valid two character State code or a valid destination depending on the message type. Multiple destinations may be specified, in which case they must be separated with a comma. The entire field must always be terminated with a period.

CTL-FLD The control field is an optional field. If present the first character must be an asterisk and must be followed by exactly 10 additional characters. If present the field must be terminated with a period, if not present nothing must be sent.

TXT This is an optional field and is not required.

TEXT This is the actual text of the message, it is normally composed of Message Field Code descriptors and their associated data. Each field except the last must be terminated with a period.

Examples:

[header.]RQ.IN0000000.IL.*1234567890.LIC/ABC123.LIY/2000.LIT/PC

Where:

RQ	Message key - registration query for out of State
IN0000000	The ORI
IL	The destination State
*1234567890	The optional control field
Text	The text of the message comprises the remainder of the message

The NCIC 2000 Format

These messages will include all of the valid NCIC and In-State Hotfile MKE's. They will be routed to the proper destination even though a destination field is not provided. As is the case with NLET formatted messages, these messages can also generate transactions to other data bases depending on the specific MKE configuration. All valid State Hotfiles database and NCIC 2000 transactions will follow this format. The format of the messages are described below:

[header.]MKE.ORI.text

Where:

MKE The MKE field must be a valid NCIC message key, if not an error message will be returned to the input station indicating 'invalid MKE'. In addition, the incoming session will be checked to see if it has the authority to execute the message key. If not an error message will be generated back to originating station indicating 'not authorized to use MKE'.

ORI The ORI must be the valid State ORI assigned to the input station. If the ORI is not valid or if it is not assigned to the originator an error will be returned.

Text The text portion of the message keys will comply with NCIC standards.

Example:

[header.]QV.IN0000000.LIC/ABC123.LIS/IN

Where:

FOXY The DAC name
QV The NCIC message key - query vehicle
IN0000000 The ORI assigned to the input station
Text The remainder of the message is text: LIC/ABC123.LIS/IN

Output Message Formats

This section will describe output message formats to any stations utilizing the DMPP 2020 protocol.

The processing header will be used by the receiving system to perform processing related to grouping and displaying the messages. The format of the processing header is as follows:

TDAC.Reference.Source.MKE.Date/time.

Where:

TDAC	Is the name of the device to which the message is addressed
Reference	Is a 10 character alphanumeric field as entered with the original request, it will contain the constant UNKNOWN for responses for which the value is not known or the constant UNSOL if the message is an administrative message.
Source	This field will consist of the OpenFox™ mnemonic associated with the source of the message.
MKE	This is the response MKE, wherever possible this will be the same as the request message key, AM for administrative messages, SM for notifications, ER for error notices.
Date/Time	This will be the date and time that the message is being output by the switch, it will have the format YYYYMMDDHHMMSSxx

Where:

YYYY	The 4 character numeric year
MM	Two character numeric month
DD	Two character numeric day
HH	Two character numeric hour (military format)
MM	Two character numeric minute
xx	Two character numeric field representing hundredth of a second